Software Engineering Institute

Best Practices for Artifact Versioning in Service-Oriented Systems

Marc Novakouski
Grace Lewis
William Anderson
Jeff Davenport

January 2012

TECHNICAL NOTE
CMU/SEI-2011-TN-009

Research, Technology, and System Solutions Program

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work funded and supported by the United States Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the

SEI Administrative Agent
ESC/CAA
20 Schilling Circle, Building 1305, 3rd Floor
Hanscom AFB, MA 01731-2125

Table of Contents

List of Figures

List of Tables

Abstract

This report describes some of the challenges of software versioning in a service-oriented architecture (SOA) environment and provides guidance on how to meet these challenges by following industry guidelines and recommended practices. Managing change in software systems becomes more difficult as the software increases in size, complexity, and dependencies. Part of this task is software versioning, in which version identifiers are assigned to software artifacts for the purpose of managing their evolution. However, software versioning is not a self-contained task. Versioning decisions affect a wide range of processes that fall under the broad heading of change management. With the advent of SOA as a software-development paradigm, software versioning has become even more entwined with the software life cycle, mainly due to the highly distributed nature, multiproduct outcome, and multilayer implementation of service-oriented systems. The report describes typical items that a versioning policy for a service-oriented system should contain, including which artifacts to version, how to apply version control, and the impact of versioning on each phase of the life cycle within an SOA infrastructure.

1 Introduction

Managing change in software development is complex. Software configuration management (SCM) is the discipline of managing change in software work products. In the Carnegie Mellon® Software Engineering Institute (SEI) Capability Maturity Model Integration® (CMMI®) framework, SCM is defined as "establishing and maintaining the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits" [CMMI Product Team 2010].[1] Software versioning—the process of assigning unique names or numbers to specific software artifacts—is part of SCM.

Developers typically describe software versioning as a self-contained set of practices that involve selecting and applying a paradigm for generating the unique names or numbers assigned to software artifacts. These numbered artifacts help developers manage them as they evolve. However, software versioning has become increasingly entangled with the processes of life-cycle management, testing, and interorganizational governance, especially in service-oriented systems development [Lhotka 2007, Lublinsky 2007b, Schepers 2008, Juric 2010].

We define *service-oriented architecture* (SOA) as a way of designing, developing, deploying, and managing systems, in which

- services provide reusable business functionality via well-defined interfaces
- there is a clear separation between service interface and service implementation—and service interface is just as important as service implementation
- service consumers are built by using functionality from available services
- an SOA infrastructure enables discovery, composition, and invocation of services
- protocols are predominantly, but not exclusively, message-based document exchanges [Lewis 2009]

We define a *service-oriented system* as a system that is built using the SOA paradigm. Figure 1 presents a high-level notional view of a service-oriented system. This figure shows the main four elements of a service-oriented system: service consumers, SOA infrastructure, service interfaces, and the interfaces' associated service implementation.

[1] ® Carnegie Mellon, Capability Maturity Model Integration, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

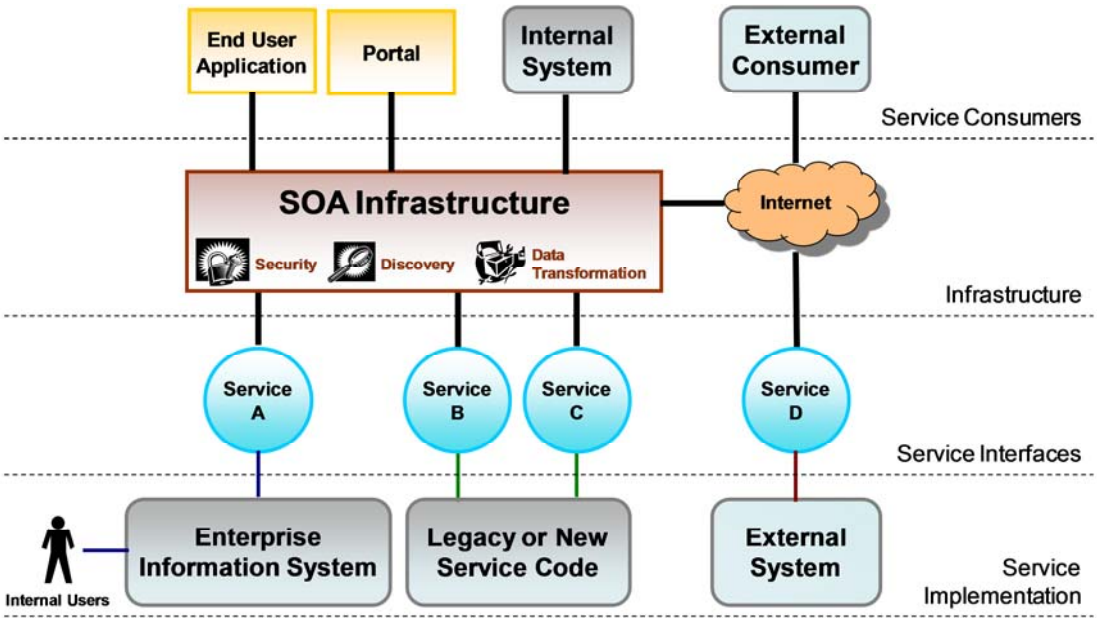

Figure 1: High-Level Notional View of a Service-Oriented System

In service-oriented system development, software versioning is an essential process for reducing the impact of change on service consumers, reducing maintenance costs, and improving the overall manageability of services [Peltz 2004, Lublinsky 2007a, Schepers 2008]. However, software versioning tends to be much more difficult when dealing with service-oriented systems, mainly because of the distributed nature of SOA development [Peltz 2004]. It is common for different groups, from either the same organization or different organizations, to develop service-oriented systems collaboratively [MacKenzie 2006, Lublinsky 2007b]. As a result, developers must address several software versioning challenges that are unique to the distributed-development paradigm.

The first challenge of versioning service-oriented systems is lack of centralized control. When software is developed in a distributed manner across a number of groups or organizations, the control that a single SCM group usually exercises is instead distributed over multiple groups, potentially within multiple organizations. As a result, developers must adapt versioning processes that require centralized control to account for communication and coordination among different SCM groups.

The second challenge of versioning service-oriented systems is the ripple effect of change. Because each organization controls only part of the service-oriented system, its members must strictly coordinate release schedules to ensure compatibility. As a result, even small changes to services can have a significant impact on service consumers, which ultimately affects the service provider as well [Lublinsky 2007b]. This is because all service consumers rely on both the stability of the service interface and the expected quality of service (e.g., performance, availability, or reliability).

The third challenge of versioning software in service-oriented systems stems from the separation of the service interface from service implementation, which is often referred to as *opacity*. Ideally, services provide an abstract service interface that any service consumer can use without concern for service-implementation details [Laskey 2008]. In reality, however, different service consumers

have different needs from a functional or quality-of-service (QoS) standpoint. This challenges developers to achieve a balance between supporting the fundamental service-oriented tenet of reuse through opacity and meeting different service consumers' needs based on business drivers. These balancing activities affect design decisions critical to service development, such as the number of interfaces and protocols to support for the same service [Laskey 2008].

We have two goals for this report on the recommended practices for artifact versioning in service-oriented systems. First, we inform policy makers, software managers, and engineers responsible for designing and implementing configuration-management policies for service-oriented systems about the challenges related to software versioning. Second, we provide general guidance, recommended practices, and further resources to deal with these challenges. We organized the report as follows: Section 2 divides the topic of versioning in SOA environments into what to version and how to version, and Section 3 covers how versioning affects service life-cycle management practices. Section 4 provides a summary of related work. Finally, Section 5 summarizes our findings.

2 Versioning Policy for Service-Oriented Systems

Any versioning policy for a software-development effort must answer two fundamental questions:

1. What to version: What artifacts should we place under version control?
2. How to version: What tools, techniques, and naming schemes should we use to implement version control?

The following sections provide guidelines on how to answer these two questions. This work extracts the fundamental concepts relevant to SOA versioning from our experience and the body of related literature and uses these concepts to build a set of relevant guidelines and practices that we summarize throughout the text as recommendations. These practices, guidelines, and recommendations address issues that come from using specific SOA technologies as well as challenges that stakeholders must address when developing service-oriented systems, regardless of the technology used. To assist the reader, we list the recommendations found in Sections 2 and 3 in the appendix for easy reference.

2.1 What to Version

Selecting the artifacts to version in a traditional software-development effort is usually relatively straightforward, and there is a great deal of guidance for doing so [CMMI Product Team 2010]. For service-oriented systems, however, "what to version" must include concepts, agreements, and configuration elements that are not traditionally versioned. Therefore, developers need to consider issues such as managing different QoS needs, complying with consumers' needs for services that are in different stages of development, and managing different interface expectations [Lublinsky 2004]. This section provides an overview of considerations to use in selecting which software artifacts to version.

2.1.1 Key Artifacts

SCM includes identifying the work products that become configuration items, and thus part of configuration baselines [CMMI Product Team 2010]. In the following sections, we discuss the key artifacts that are often part of service-oriented systems.

2.1.1.1 WSDL Documents

A Web Service Definition Language (WSDL) document is a software artifact in Extensible Markup Language (XML) format that describes a web service [Christensen 2001].[2] This might include the location of a service (i.e., service endpoint), its capabilities (i.e., operations), and the QoS it provides. Because the WSDL document typically includes most, if not all, of the information nec-

[2] Although there are many options for service implementation, the most common implementation is WS-* web services. WS-* web services (1) represent data in XML; (2) describe service interfaces in WSDL; (3) transmit payload with Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP); and, optionally, (4) use Universal Description, Discovery, and Integration (UDDI) as the directory service. In addition, although they are not part of the basic implementation, more than 100 standards exist to support other system qualities, such as WS-Security for security and WS-ReliableMessaging for reliability. Most of the guidance in this report is specific to WS-* web services, but it can be generalized to include service implementations.

essary for a service consumer to interact with the service, designers often consider it the first and most important software artifact to version when developing web services [Brown 2004, Evdemon 2005, Poulin 2006, Fang 2007, Leitner 2008, Juric 2010]. However, directly versioning the WSDL document has some drawbacks. Developers must account for issues such as granularity, opacity, and method signature [Poulin 2006, Lublinsky 2007b], and we discuss these elements in Sections 2.1.2 and 2.1.3.

Recommendation 1: Place all WSDL documents under version control.

2.1.1.2 XML Schemas

An XML schema defines the structure and type of content included in XML documents. Developers typically use it to describe the structure and content of messages exchanged with web services [Lublinsky 2007a]. For example, designers commonly describe data types in XML schemas and then reference them in WSDL documents to define operation parameters, instead of defining the data types directly inside the WSDL documents.

In SOA environments, XML schemas are key artifacts for developers to place under version control [Peltz 2004]. Some experts propose that the XML-schema representation of web-service messages should be the basis of service versioning, instead of WSDL documents [Evdemon 2005, Lublinsky 2007b]. We examine these issues further in Section 2.1.2.2.

Recommendation 2: Define data types used in service interfaces in separate XML schemas, and place them under version control.

2.1.1.3 BPEL/Web-Service Composition and Orchestration Documents

Individual web services are often composed into complex business processes. While several technologies will support composing and orchestrating web services, the development community regards Business Process Execution Language (BPEL) as a de facto standard for composing individual web services into business processes [Cobban 2004, Juric 2010]. BPEL process documents are key artifacts in policies for web-service versioning, both as artifacts to version and as artifacts that are affected by versioning.

Because service-composition documents define how services interact when they execute a larger business process, they should include the version information associated with the services that they execute [Juric 2009a, 2010]. Business processes also evolve. For example, as organizations bring on third-party contractors and suppliers, they must update documents for business-process composition to support new processes and also deal with any potential requests by consumers of the old processes, which may or may not still be supported. This is especially true in the case of long-running business processes.

Recommendation 3: If the development effort for service-oriented systems includes composite services, then consider the documents that control the composition as key artifacts and place them under version control.

2.1.1.4 Runtime Infrastructure

Service-oriented systems often rely on an SOA infrastructure such as an enterprise service bus (ESB) to act as the mediator between consumers and services [Fulton 2009]. Changes or upgrades in technology or changes in the system's quality attribute requirements can trigger changes in components of the infrastructure, especially when a project is in its early stages. Examples of changes include the type of message transport, message encoding, and addressing schemas [Lublinsky 2007b]. The standards implemented by infrastructure components can also change. If a particular project is likely to have infrastructure changes, developers should apply techniques such as those described in Section 2.2.2.5 to manage versioning of the SOA infrastructure.

Recommendation 4: Ensure that the versioning policy contains guidance for how to handle changes in SOA infrastructure components.

2.1.1.5 Metadata Documents

WSDL documents can contain different types of metadata to describe a service, but they might not include metadata that is important for specific implementations of service discovery, such as QoS, service rating, and associated test cases. For example, QoS is usually specified in service-level agreements (SLAs), which are not part of the WSDL documents. In this case, developers should maintain additional metadata documents that consumers can use to find the services that best meet their needs when engaged in processes such as dynamic discovery, in which the best match for a service request is determined at runtime [Hagge 2001].

Recommendation 5: Identify all metadata that is relevant to service consumers, decide how to document it, and place the resulting artifacts under version control.

2.1.2 Service Interfaces

Two issues important for defining service interfaces are interface opacity and interface-operation types. The decisions made regarding these issues define what artifacts of service-interface description are versioned.

2.1.2.1 Interface Opacity

Opacity refers to the amount of information about a service that is exposed to consumers. A fundamental principle of service-oriented design is that services should hide implementation details from consumers [Lublinsky 2004, Laskey 2008]. Developers often accomplish this by providing well-defined, generic interfaces that serve multiple purposes [Brown 2004]. In this type of implementation, the runtime environment often handles all versioning details by using techniques such as a service broker (see Section 2.2.3) [Leitner 2008]. Even if there are multiple versions of the service, consumers see a single service interface. However, this approach may not be appropriate for consumers who want more transparency into details such as the QoS of a specific service operation [Leitner 2008]. Some consumers also have an aversion to change, so they prefer to rely on established, verified versions of a service rather than to upgrade [Lublinsky 2004]. In this case, the provider should expose multiple versions of the same service so that consumers can find the version that best fits their needs [Lublinsky 2004, Peltz 2004].

The service provider must find a balance between service-interface opacity and transparency that best meets the needs of both current and potential consumers [Laskey 2008, Juric 2010]. This decision affects versioning strategies, such as how many service versions to support and for how long. For example, a large number of consumers with few tailoring needs suggests high opacity. Many tailoring needs or aversion to change in a small consumer base suggests high transparency. A mix of needs requires further analysis to find the right balance.

Recommendation 6: If the service provider will support multiple interfaces for a single service, include policies about how long to support each exposed interface. In addition, use a naming convention that indicates that these are all variants of the same interface.

2.1.2.2 Interface-Operation Types

In traditional programming, functions or methods have a defined name with a defined input type that makes each function or method entirely unique. However, a technique called *overloading* allows a function or method to accept and process different types of input [Watt 1993]. Developers accomplish overload by creating different instantiations of the function or method with the same name, but differentiating them by the type of input they accept. From the perspective of the caller, there is only one method that can receive different types of input.

Operations in a service interface can also be set up in both ways. Each operation may be unique in terms of name and input type, or the operation may be set up to accept generic input that it can then decide how to process [Evdemon 2005]. The latter approach, in which a service operation accepts a generic XML document as input that is then parsed to determine how to process the document, is often called *semantic messaging*, and it leverages the standard SOAP encoding of web-service messaging to achieve flexibility in inputs [Lublinsky 2007b]. If this is the case, then XML schemas that represent the type of operation input are key artifacts for versioning the web service.

Additionally, the design decision that determines how to define operations (either unique or overloaded) influences the versioning strategy, similar to the design decision about opacity. For example, if consumers require simple, fine-grained services with limited input that does not change often, then unique interfaces are typically a good choice. Developers should use overloaded interfaces for coarse-grained services that receive complex input or evolve quickly.

Recommendation 7: Develop policies for how long to support multiple versions of the same interface that account for the faster change rates of unique interfaces. Even though overloaded interfaces have a lower change rate, place both the interface itself and the schemas that represent operation input types under version control.

2.1.3 Versioning Granularity

In an SOA environment, versioning granularity is related to the concept of service granularity:

> *Granularity is a relative measure of how broad a required piece of functionality must be in order to address the need at hand. Fine-grained services address small units of functionality or exchange small amounts of data. Consequently, to build complex business processes, companies would have to orchestrate large numbers of such services to effectively automate*

the process—a difficult, often Herculean task. Coarse-grained services, however, encapsulate larger chunks of capability within a single abstracted interface, reducing the number of service requests necessary to accomplish a task, but on the downside, they might return excessive quantities of data, or it might be difficult to change them to meet new requirements. As a result, an architect must craft the right balance of fine-grained and coarse-grained services to meet the ongoing needs of the business. [Schmelzer 2006]

As Schmelzer indicates, selecting an appropriate service granularity is one of the most important decisions in designing services.

Similarly to defining service granularity as "units of functionality" [Schmelzer 2006], we can define *versioning granularity* as the "units of versioning" selected for a specific artifact or set of artifacts [CMMI Product Team 2010]. We determine for a given artifact whether to assign version numbers to the entire artifact, to the individual elements that make up the artifact, or to some other combination. In service-oriented systems, this translates to versioning the entire service (WSDL documents, XML schemas, and all associated artifacts) or to versioning the individual operations that are part of the service [Poulin 2006, Lublinsky 2007b]. Some sources even suggest versioning the individual components of the operations, such as description, interface, and endpoint [OASIS 2004]. But as Peltz and Anagol-Subbarao warn, "[I]t can be appealing to version down to the very lowest levels in accordance with object-oriented design principles, however doing so may not necessarily be in the best interests of supporting a consistent versioning policy" [Peltz 2004].

Recommendation 8: Consistently with standard software development, version all key artifacts for internal use. From the perspective of the external service consumer, version either the entire service or the individual operation interfaces, depending on the needs of the potential consumers. Avoid exposing version information at a level lower than operation, as that is likely to confuse service consumers.

2.1.4 Quality of Service and Service-Level Agreements

As we briefly mentioned in Section 2.1.1.5, developers often make additional metadata describing a service available to service consumers [Evdemon 2005]. QoS, as described in an SLA, is one of the most important pieces of metadata for service consumers to access [Poulin 2006, Lublinsky 2007b, Leitner 2008]. A common strategy that service providers use—especially third-party service providers—is to offer different tiers of quality according to an SLA between the service provider and service consumer [Lublinsky 2004]. These strategies typically enable different QoS levels through different service-method interfaces or through specification in the input message to the service (as described in Section 2.1.2.2). Alternatively, QoS can be part of the service interface that is invisible to consumers, thus resulting in a single interface to the service that does not specify QoS. As a result, a decision to offer different levels of service affects how developers will expose and version the services.

Recommendation 9: If a system provides services at different QoS levels, each with a different service interface, place all exposed interfaces under version control. In addition, use a naming convention that indicates that these are all variants of the same interface.

2.2 How to Version

Once designers have decided what to version, the decisions about how to apply version control are fairly standard, regardless of the type of software being developed. These decisions concern naming schemes, techniques for applying and managing version identification, and tools for coordinating the version-control decisions. However, there are additional considerations for building service-oriented systems. This section provides an overview of naming schemes, techniques, tools, and standards that are appropriate for SOA environments, as well as additional considerations and recommendations for developing a comprehensive versioning policy. We also discuss ongoing research for readers interested in the evolution of SOA versioning.

2.2.1 Naming Schemes

Naming schemes are processes and conventions used to assign a software artifact (or group of artifacts) a specific name, and they are the backbone of a good versioning policy. Choosing a good naming scheme will simplify the management of releases for the service provider and will clearly communicate important information, such as features and QoS, to potential consumers [Peltz 2004]. However, no single naming scheme is ideal for all SOA environments. The appropriate naming scheme for a project will depend on a number of factors, including decisions about what constitutes a new version of a given artifact, how the developers define change types, and what kind of web-service technologies they use. In the following sections, we discuss these factors and provide sample naming schemes.

Recommendation 10: Construct a comprehensive naming scheme, including version creation thresholds, change types, compatibility rules, and a scheme for determining the identification name or number of a new artifact.

2.2.1.1 Creating a New Artifact Version

The first decision in putting together a versioning policy is to define what kinds of changes result in the creation of a new version of an artifact. Lublinsky offers a good rule of thumb:

> *Any change in the service, whether it is a change in the interface or implementation that might impact consumer execution, should lead to creation of the new version of service (method). [Lublinsky 2007b]*

Developers need to remember that, if they make changes to the implementation that could affect service consumers, they should create a new version of the service or operation. This is good practice even if they use a degree of service opacity that favors hiding implementation details and even when there are no apparent changes to the exposed interface [Laskey 2008]. Creating a new version is necessary because of assumptions that the consumer might have about a given service invocation, such as QoS and pre- and post-conditions [Lublinsky 2007b].

However, developers can make several kinds of changes to a service that would not necessarily require a version change. Laskey provides a list of changes to a service description that would not alter the way in which the service interacts with consumers [Laskey 2008]:

- *correcting errors that do not significantly change the description (e.g., a simple typo)*

- *correcting errors that do significantly change description (e.g., the word NOT was missing from the functionality description)*
- *adding information (e.g., an additional real-world effect that was previously considered inconsequential)*
- *removing information that was previously required or thought useful (e.g., the number of times the service has been used)*
- *consolidating elsewhere the specifics of some information and replacing the occurrences in the service description by a link to the consolidated location (e.g., version history)*

2.2.1.2 Change Types

Defining a naming scheme for any software-development effort requires defining the types of changes that lead to new versions of specific artifacts, groups of artifacts, or entire applications. Deciding what those changes are, and how to classify them, is a key aspect of any versioning policy.

There are several ways to classify change types. The most common classification scheme is based on compatibility with other versions of the software.[3]

Compatibility. The types of changes in development efforts for service-oriented systems are typically separated into two groups: backward compatible and non-backward compatible. There are many ways to define backward compatibility. One definition of *backward compatibility* for services is as follows:

> *A service is backward compatible if it is compatible with earlier versions of itself—especially if these earlier versions are expected to be deprecated. A key design goal of backward compatibility is to ensure that consumers of older versions of a service can continue to interoperate with the new version. Backward compatibility focuses on preserving existing service contracts while extending the service to add new features. [Evdemon 2005]*

In the context of services, backward compatibility means that a new version of a service can accept input and generate output in a way that is equivalent to the old version, from both a functional and a QoS perspective.

A simple example of a change that is not backward compatible is changing a service interface (an operation) from accepting a floating point number representing dollars to accepting an integer number representing cents for greater accuracy [Brown 2004]. This syntactic change to the interface may not cause execution of the service to fail because many modern systems can convert from float to integer on the fly. However, the consumer of the service will not receive the expected return value because of the change in the input, which demonstrates that backward-compatibility issues can result in execution failures as well as logic errors. Consequently, an organization must communicate with service consumers about all changes to a service that are not backward compatible.

Because of service opacity, ensuring backward compatibility is a common goal when developing services, but this is not always possible. Therefore, when designing or selecting a naming scheme,

[3] Leitner and colleagues describe an alternative classification scheme that breaks down change types into three categories: nonfunctional changes, interface changes, and semantic changes [Leitner 2008].

developers can facilitate continuity of service by defining what types of changes are backward compatible and what types are not. For web services, backward-compatible changes to service interfaces include adding a new operation, creating a new interface for an existing operation, and creating and adding a new XML schema to a WSDL document [Brown 2004, Evdemon 2005]. Some examples of non-backward-compatible changes are [Brown 2004, Evdemon 2005]

- removing an operation
- renaming an operation
- changing the parameters (in data type or order) of an operation
- changing the structure of a complex data type
- changing the content or structure of a response message (other than type restrictions)

Developers should keep in mind that implementation changes can also affect the usability or performance of a service, even if the interface itself does not change [Laskey 2008]. For example, a service used by the Department of Defense (or any other organization that has preapproved or cleared product lists) might change the underlying database technology to a technology that is not approved for use. In this case, the interface, function, and QoS of the exposed service may be identical to previous versions, but from the perspective of the service consumer, the service is unusable.

In addition to backward compatibility, it is also important to consider forward compatibility:

> *Forward compatibility means that newer versions of producers (services) can be deployed without breaking existing consumers. Forward compatibility is much harder to achieve than backward compatibility since the service must be expected to gracefully interact with a number of unknown or unexpected features. Forward compatibility can be compensated for by adopting standards designed to ignore unrecognizable elements that may appear in the future. This concept (sometimes referred to as MUST IGNORE) was a key design goal in the W3C XML standard. [Evdemon 2005]*

Ensuring forward compatibility is a useful goal, especially in an environment where services undergo constant change. By designing a forward-compatible service, developers can avoid future maintenance costs, thus reducing overall costs for the life of the service. However, this type of forward compatibility can be extremely difficult to implement because it means that developers must ensure compatibility with an unknown future version of the service. Therefore, no one can truly guarantee that a service will be forward compatible [Laskey 2008]. Although developers should think about forward compatibility when composing a versioning policy, it is unlikely to be a key goal of the policy.

Recommendation 11: Describe backward- and forward-compatibility requirements and goals in the versioning policy, as appropriate to the context. Actively seek backward compatibility in service-oriented systems development, but realize that forward compatibility is much more difficult to ensure and may not be feasible.

Classification. A common way of classifying backward-compatible and non-backward-compatible changes is as *minor* or *major* changes. Minor changes are typically considered backward compatible, while major changes are typically not considered backward compatible [Peltz

2004, Lublinsky 2007]. An expansion of this scheme can include *revisions*, or changes that do not affect the functionality of a service or the experience of a service consumer [Lublinsky 2007]. Developers typically build these change types into the naming scheme, such as in this example:

> *When you are sequentially naming your version you can use a convention of vMajor#.Minor#/SERVICE_NAME, where `Major#` is the major version release and Minor# is the minor number release. [Peltz 2004]*

In this way, the definition and subsequent classification of the types of changes that developers can make to a service (and to its comprising software artifacts) drive the naming scheme of a versioning policy.

Recommendation 12: Use major and minor version classification to communicate compatibility issues to service consumers. However, do not make it a critical part of a versioning policy in SOA environments because it is not appropriate for all contexts.

Merging. Merging is one type of change that developers seldom plan for in versioning policies. Typically artifacts evolve separately, but some situations require merging [Laskey 2008]. It may rarely be relevant to cover this situation in a naming scheme, but it will be important in those situations where merging occurs.

2.2.1.3 Typical Naming Schemes

There are multiple sources for typical naming schemes in SOA environments:

- Anagol-Subbarao and Peltz provide service naming schemes, including a major/minor scheme as well as one that uses time stamps [Peltz 2004].
- Poulin suggests a "compound version identifier" that accounts for many weaknesses of typical naming schemes within the SOA environment [Poulin 2006].
- Juric and others propose a BPEL-based naming scheme [Juric 2009a, 2010].
- At a more theoretical level, Conradi and Westfechtel present a scientific analysis of common versioning models [Conradi 1998].
- Wikipedia contains a summary of common schemes and can be a useful introduction to the topic [Wikipedia 2011].

Recommendation 13: Use basic numeric naming schemes, including major/minor designations, for web services. Consider more complex schemes as the consumer base or capabilities increase in size.

2.2.2 Artifact Versioning Techniques

There are multiple techniques and strategies to use in versioning specific types of artifacts. This section provides recommended practices and references for the primary artifacts that are important to place under version control in an SOA environment.

2.2.2.1 WSDL Documents

A wide range of techniques are available to support the versioning of WSDL documents:

- Brown and colleagues provide examples for how to create and leverage unique WSDL namespaces to enable versioning [Brown 2004].
- Evdemon offers specific strategies for how to evolve WSDL documents to maintain backward compatibility [Evdemon 2005].
- Juric and colleagues present a detailed analysis of extensions to WSDL that could support versioning [Juric 2009b].
- Fang and colleagues provide examples of what a version-aware WSDL extension could look like [Fang 2007].
- A more complex solution could involve the concept of a semantically annotated WSDL, such as SAWSDL [W3C 2007, Leitner 2008].

Evdemon also contributes the following general design principles [Evdemon 2005]:

- ***Design Principle #4:*** *When adding new data structures, make them optional, and add them to the end of service request messages.*
- ***Design Principle #5:*** *Changing service response messages (other than type restrictions) are breaking changes, and will require a new version of the service.*

Recommendation 14: Use WSDL documents as the backbone of any service versioning strategy, and use the namespace field to differentiate services and interfaces. In more complex environments, extend or annotate the WSDL format to manage extra information.

2.2.2.2 XML Schemas

SOA versioning literature also includes a wide range of techniques for versioning XML schemas. Evdemon provides three strategies, with examples, for introducing versioning information into XML schemas [Evdemon 2005]:

1. *Use a new XML namespace for major version releases.*
2. *Keep XML namespace values constant and add an XML-schema version attribute.*
3. *Keep XML namespace values constant and add a special element for grouping custom extensions.*

In addition, Evdemon proposes multiple design principles that ease the process of versioning XML schemas [Evdemon 2005]:

1. *Use **targetNamespace** to communicate major version releases.*
2. *Judicious use of unambiguous wildcards can help minimize service versioning.*
3. *Extensions must not use the **targetNamespace** value.*

Peltz and Anagol-Subbarao provide additional guidance on how to implement these solutions in XML schemas as well as a cost/benefit analysis, examples, and possible hybrid solutions [Peltz

2004]. They also suggest the industry-standard practice of using XSL Transformations (XSLT) to translate between versions of XML schemas.

Finally, Lublinsky provides four versioning methods for XML schemas, including a final, hybrid solution that consists of [Lublinsky 2007]

- [*componentizing*] *the overall schema in logical partitions using multiple namespaces, thus containing changes*
- *defining a new namespace (reflecting the major version information) for every major version of each schema*
- *denoting every minor version as a schema version in a major version namespace. Because minor versions are backward compatible, generated marshaling/unmarshaling code also will be backward compatible*

Recommendation 15: Make service design decisions, particularly regarding the use of namespaces, before defining XML-schema versioning policies.

2.2.2.3 BPEL Specifications

The BPEL specification currently does not include versioning support. However, some research has been done on how to extend BPEL to have versioning capability [Juric 2010]. Juric and colleagues have performed detailed work in this area that developers may adopt if necessary [Juric 2009a].

Recommendation 16: For composite services, version the composition-control documents and make them version aware. With BPEL, use the extensions described by Juric and colleagues [Juric 2009a]. Otherwise, consult recommended practices for the selected business-process-engine technology to enable version awareness.

2.2.2.4 Service-Level Agreements

The WSDL specification of services sometimes includes QoS metrics. However, this is not always true, and when it is not, there is no standard way of capturing and exposing this information. In addition, service-level QoS metrics usually focus on performance aspects, such as response time, but there are other QoS parameters of interest, such as availability (e.g., uptime) and security (e.g., type of encryption), that developers should capture for services.

Work by IBM to create the Web Service Level Agreement (WSLA) language specification has begun to address this problem, and developers should consider it for SOA environments that offer multiple QoS levels [Ludwig 2003]. Bianco and colleagues present the state of the practice in SLA specification, along with guidelines from a quality attribute perspective [Bianco 2008].

Recommendation 17: Place SLA documents under version control. Either use an ESB infrastructure to provide a standard way of managing SLA concerns, or investigate custom solutions for SLA management.

2.2.2.5 Runtime Infrastructure

Few researchers have investigated how to manage change in runtime infrastructures from a versioning standpoint. Lublinsky suggests the following:

> *. . . it is always desirable to implement "backward compatibility," ensuring that new infrastructure can "understand" and support messages produced by the old infrastructure and produce messages [that are] compatible with it. In reality, it might be too expensive, or even technically impossible. Moving all of the existing service implementations and consumers to the new infrastructure is typically fairly expensive and time-consuming, requiring versioning support that provides interoperability between two different service infrastructures. The most popular solution to this problem is a service adapter . . . [in which] the invocation [from the service consumer] goes through the adapter mediating between the service infrastructures. From the point of view of the service consumer, the adapter acts as a service provider. The adapter then invokes the [actual] service provider acting as a service consumer. [Lublinsky 2007b]*

Recommendation 18: Plan the service infrastructure well in advance to avoid significant infrastructure change. If possible, use common open-technology standards to minimize the potential impact of infrastructure change. Architect service-oriented systems in a way that allows the infrastructure to evolve with minimal disruption to services and consumers.

2.2.3 Using a Service Broker/Router

There are two basic architecture patterns (or integration approaches), one direct and one indirect, for connecting a service consumer to a service provider [Lublinsky 2004]:

- ***Direct Routing:*** *In this architecture, the service consumer obtains the endpoint of the service and connects directly.*
- ***Intermediary-Based Routing:*** *In this architecture, the consumer never communicates with the service directly. The intermediary detects the client making the request and retrieves policy files for the particular consumer. The intermediary then invokes the appropriate service based on the client request and any applicable internal policy, such as quality of service or deprecation issues.*

Each approach has several advantages and disadvantages [Lublinsky 2004]. However, from a versioning perspective, researchers most often recommend the indirect approach of using a service router or broker [Brown 2004, Peltz 2004, Evdemon 2005, Lublinsky 2007, Leitner 2008, Juric 2010]. This approach is balanced between service transparency and opacity because the broker can select the appropriate version of the requested service for the given service consumer, and it hides capability, QoS, and deprecation issues from the consumer [Leitner 2008]. Poulin refers to this broker capability as a “Version Control Policy” [Poulin 2006]. While it does introduce additional overhead, as well as complexity on the service-provider end, it adds an element of control for the service provider in an SOA environment.

Service brokers have additional benefits for versioning. Brokers are preferred, if not required, for use with service composition and orchestration languages, such as BPEL. In this case, developers must fully leverage the extensions discussed by Juric and colleagues [Juric 2009a]. In addition, brokers enable exposure of the design of more coarse-grained services to consumers, which can

simplify the task of versioning exposed services [Peltz 2004]. This is accomplished by placing the broker between the interface and the service logic, and decoupling the interfaces from the actual services, as in the web-service façade pattern. This allows the designer to create a closer match between the exposed services and the business processes that they support and results in the exposure of coarser grained services that are easier to maintain because they require less communication and coordination with external consumers [Erl 2009].

Another advantage of service brokers is the way they manage requests for deprecated services [Brown 2004]. If a service is deprecated, a service broker can translate the request into a form that it can send to the version of the service that most closely matches the functionality and quality provided by the requested deprecated service. In more complex systems, this might require additional action, such as notifying the consumer that the requested version of the service is unavailable [Evdemon 2005]. This is often a better solution than a server failure, which is the only option if the consumer attempts to access the service directly. The best solution is system and consumer dependent.

One drawback of using service brokers is that they introduce additional overhead. This can be a problem if the service infrastructure has difficulty providing the advertised QoS. For example, if a service broker provides indirect access to five different versions of the same service (each of which advertises different QoS levels in its SLA), then the overhead introduced by the service broker must not violate the most stringent SLA of the supported services [Lublinsky 2007b]. In such a situation, designers might use different brokers for groups of services at similar advertised QoS levels, or they might use direct routing for the services with the most stringent QoS requirements.

If an SOA development effort decides not to use service brokers for this or for other reasons, then one common approach for managing multiple versions of the same service is to publish different versions of the same service under different endpoint URLs [Juric 2010].

Recommendation 19: In all but the most basic service-oriented systems (i.e., those with a small number of services that have well-known consumers), use some form of broker or router to simplify the interface exposed to consumers and enable greater control by the provider.

2.2.4 Tools for Versioning

Many tools are available to support versioning in an SOA environment. We can separate these tools into those that assist during the development phase and those that assist during runtime.

2.2.4.1 Development Tools

Two main types of tools can support versioning during the development phase: version-control systems (VCSs) and integrated development environments (IDEs).

VCSs manage the creation and evolution of artifacts. It is critical for all development efforts of any size to adopt a VCS, whether it is a service-oriented development effort or not. Examples of popular VCSs include CVS [CVS 2011], Subversion [Apache Software Foundation 2011], Perforce [Perforce 2011], Clearcase [IBM 2011], and Git [Hamano 2011].

IDEs are designed to make programming easier by providing a user-friendly interface that automates many manual programming tasks, such as constant recompilation. Examples of widely used IDEs include Eclipse [Eclipse Foundation 2011a] and NetBeans [NetBeans 2011]. IDEs are not critical for versioning but are helpful because some IDEs integrate well with several VCSs. For example, Subversion integrates well with the Eclipse IDE. Also, some IDEs include support for service-oriented systems development. For example, one version of the Eclipse IDE is the Eclipse SOA Platform, which specifically supports SOA development [Eclipse Foundation 2011b]. IDEs tend to be good choices for service-oriented systems development, beyond the versioning aspect. A coordinated approach to development tools can simplify the development process; therefore, developers should select VCSs and IDEs in concert.

Recommendation 20: Select a VCS that is sufficiently robust to accommodate all the needs of the software development project. To improve productivity, also select an IDE that integrates well with the chosen VCS.

2.2.4.2 Runtime Tools

The primary tool that developers use to support versioning at runtime, aside from a broker, is a service registry [Brown 2004; Evdemon 2005; Fang 2007; Lublinsky 2007b; Leitner 2008; Schepers 2008; Juric 2009b, 2010]. Service registries[4] store service information, such as WSDL documents and XML schemas, and may be queried at runtime for discovery purposes [García-González 2009]. Service registries are also common elements of ESB products, which typically include service-broker/router functionality. By working together, service registries and brokers coordinate which services the system provides to external consumers [Lublinsky 2007b, Schepers 2008].

UDDI is the most common implementation of registries [OASIS 2011]. Because UDDI does not directly support versioning, researchers have investigated how to leverage UDDI to support versioning in SOA environments [Leitner 2008]. The main thrust of available best practices is to use the UDDI tModel.[5] Brown and colleagues discuss UDDI in detail, as well as the interaction between UDDI and WSDL documents, and offer the following recommended practices [Brown 2004]:

- *UDDI Versioning Approach 1: Advertise compliance with several interfaces.*
- *UDDI Versioning Approach 2: Introduce a version number to qualify the interface tModel.*

Evdemon expands on this approach, providing examples and these strategies [Evdemon 2005]:

- *Add a version number to the tModel.*
- *Promote compliance with multiple interfaces.*
- *Adopt a one-to-one relationship between interface versions and UDDI tModels.*

[4] Although the development community often uses the terms *service registry* and *service repository* interchangeably, we use *service registry* to describe a system element similar to a searchable directory and *service repository* to describe a system element that stores additional metadata and artifacts associated with services registered in the service registry.

[5] UDDI tModel constructs are XML-based structures that represent the interface of a web service, and UDDI registries use them to facilitate registry searches for interfaces [Yu 2005].

Juric and colleagues describe an entire model of WSDL and UDDI extensions to provide a full versioning solution [Juric 2009].

The most recent version of UDDI (Version 3) supports subscriptions [Fang 2007]. This enables native UDDI V3 registries to accept and notify subscribers of changes to services when they occur.

Recommendation 21: In all but the most basic service-oriented systems (i.e., those with a small number of services that have well-known consumers), use registries in conjunction with service repositories to store additional service metadata and related artifacts. For larger implementations, use advanced service registry features to inform service consumers of changes and deal with multiple service versions.

2.2.5 Standards

Several web-service standards will enable versioning capabilities for services. Lublinsky discusses the use of WS-Addressing and WS-Policy to specify endpoints and QoS issues, and when developers apply these protocols in concert with WSDL 2.0, they can achieve a comprehensive addressing solution [Lublinsky 2004]. Evdemon also uses WS-Addressing and WS-MetadataExchange to coordinate service selection based on metadata associated with specific versions [Evdemon 2004]. Lee provides an example of how to version messages using WS-Policy [Lee 2005]. Finally, OASIS has drafted a specification called Web Service Distributed Management: Management of Web Services (WSDM-MOWS) that attempts to address the versioning problem [OASIS 2004]. While the standard is in its early stages, service-oriented systems developers might use it to guide versioning policies.

Recommendation 22: If possible, use open web-service standards not only to support versioning if necessary but also to ensure compatibility with other systems.

2.2.6 Other Research

Enabling versioning in SOA environments is an ongoing task in the research community. Leitner and colleagues discuss a research project supporting end-to-end versioning that uses a number of different technologies and standards [Leitner 2008]. And Fang and colleagues report on a research project that addresses versioning issues from the client perspective [Fang 2007]. While this work is not mature, it provides insight into the challenge of versioning in an SOA environment.

Recommendation 23: If standard versioning approaches appear to be insufficient for a given service-oriented system, consult current research for ideas about new standards and methods for extending existing standards.

3 Service Life-Cycle Management

Service life-cycle management has a close relationship to service versioning because changes in services trigger many changes to artifacts. Peltz and Anagol-Subbarao describe a typical service life cycle [Peltz 2004]:

1. *After making the changes to a service, perform unit and functional testing of the service.*
2. *Deploy the new service through appropriate schema, WSDL, and service changes. This step might include registration to a UDDI registry or the Web services platform.*
3. *Notify the consumers of the service and pilot the new versions with one of its consumers.*
4. *Run the new and old versions in parallel for the timeframe that is allocated in the versioning plan.*
5. *Notify the consumers of the service of the date for deprecation of old versions of the service.*
6. *Remove old versions from WSDL descriptions and UDDI registries to prevent new consumers from discovering and using the older versions.*
7. *Remove functionality of the old service and add in appropriate functionality so that existing consumers are properly notified (e.g., through a SOAP fault) that the old version is no longer supported.*

In the following sections, we discuss each phase of the service life cycle—creation, deployment, deprecation, and retirement—as well as the impact of versioning on relevant processes within each of these phases.

3.1 Creation

Several processes related to creating a new artifact or a new version of an artifact are relevant to versioning, especially testing. The most important aspect of testing service-oriented systems from the versioning perspective is to include a robust backward-compatibility (and forward-compatibility, if possible) testing process. As we explained in Section 2.2.1.2, the versioning policy for developing service-oriented systems should include clear decisions on what is considered to be backward compatible and what is not. The compatibility-testing process should ensure that the service complies with the compatibility specifications that are advertised to external consumers.

It is also important to consider the needs of service consumers. If a service changes in a way that is not backward compatible with the previous version, it affects consumers who rely on the original functionality or QoS metrics of the service [Lublinsky 2004, 2007b]. Even when the interface of a service does not change at all, consumers often hesitate to switch to a new version without thorough testing [Lublinsky 2004]. One way of compensating for this aversion to change is to release a “pilot” or “early release” version of a service that allows consumers to test functionality and QoS metrics for themselves [Lublinsky 2004]. The more obvious way of dealing with aversion to version upgrades is to maintain the old version of the service until all consumers have switched. In Section 3.3, we discuss this approach in more detail.

Finally, in the development efforts of more tightly coupled service-oriented systems, designers might build a test infrastructure for services. There does not seem to be an industry-standard tech-

nique or set of tools for supporting this approach; however, some level of interorganizational testing is always recommended for multi-organizational SOA environments that produce shared services.

Recommendation 24: Align the version-control policy with the organization's testing strategy.

Recommendation 25: Use compatibility testing for new versions of a service for both backward and forward compatibility to ensure proper support for consumers.

Recommendation 26: Explicitly determine how many versions of a service to support and for how long.

Recommendation 27: Release early versions of a service to support testing by service consumers, but manage and name them consistently to differentiate clearly between test versions and production versions of a service.

3.2 Deployment

The two main aspects of deployment to consider from a versioning perspective are communication and frequency, especially in multi-organizational service-oriented systems.

Communication between organizations and their stakeholders is crucial in deploying services [Peltz 2004, Schepers 2008]. The distributed nature of service-oriented systems development can increase the complexity of coordinating service deployment, mainly due to service interdependencies that can result in significant financial impact if they are not well coordinated [Lublinsky 2007]. For example, if a business process is supported by services from multiple cooperating organizations, and if one of the organizations drastically changes its service, the entire business process would fail and negatively affect the income of all the organizations.

A good communications policy is critical to the success of multi-organizational service-oriented systems. Such a policy would ensure that all participating organizations are aware of service changes and would have sufficient details about the changes so that each organization could test against and comply with the new versions of the services in a timely manner. Developers must write communication policies that codify how communication and coordination occur with respect to new services, new versions of existing services, and the deprecation and retirement of old services, and all service providers and consumers must agree on them.

Recommendation 28: When participating in the construction of a multi-organizational service-oriented system, write a codified communication policy about service changes to ensure the smooth evolution of the system.

Communication policies for updates are typically either active or passive. The main difference between the two types of policies is the time at which a service consumer is notified of changes.

In an active communication policy, the service provider actively notifies known consumers of service version changes, either before the change occurs or at the time of the update. This is especially necessary when the exposed service or services are part of a multi-organizational business process. As mentioned earlier, one mechanism for doing this is the subscription service of the UDDI V3 registry standard, which notifies subscribers of service changes.

Alternatively, a passive communication policy either notifies users of changes at the time of service consumption (namely, when the call to the interface is made) or does not notify them at all. A passive approach might be more appropriate when a service does not have any known consumer dependencies, when the consumers are not well understood, or when service consumers have no visibility into the versions of the services that they consume. An example of a passive approach is to provide information about recent changes related to deployment, deprecation, and retirement in the response message whenever a service is called.

In some cases, it might be appropriate not to notify the consumer of changes at all, but to simply post version information to a central change-notification repository, such as a web page. This requires the consumer to actively seek out the current version information. While this puts a greater burden on the consumer to obtain current version information, it can be a useful approach for services that do not change much and expose only the most current version of a single interface. Regardless of the approach, ensuring the smooth evolution of services requires alerting consumers to versioning changes for which they may have to compensate.

Recommendation 29: Actively provide notification of changes for transparent service interfaces; use passive policies for opaque service interfaces.

Agreeing on the rate of updates—whether monthly, weekly, yearly, ad hoc, or on demand—is critical, especially in developing multi-organizational service-oriented systems. There is no "right" frequency rate for updating services; this decision is usually specific to both the organization and context and may require different update rates for different types of changes. For example, while service providers often deliver planned changes and updates on a periodic, reliable schedule, addressing issues of incompatibility or security commonly requires critical updates.

Recommendation 30: Select update rates based on customer needs and SLAs, but ensure that the procedure can accommodate on-demand changes in critical situations.

If all organizations involved in the development effort for a service-oriented system follow a combination of good update communication and update-frequency policies, they will minimize version-based conflicts during the life cycle of the system.

3.3 Deprecation

Providers of service-oriented systems have a number of reasons for deploying multiple versions of the same service [Peltz 2004, Leitner 2008]. These include supporting customers who want to use older versions of published services and SLAs that offer different QoS metrics to different customers [Lublinsky 2004, 2007b]. Service-oriented systems that take this approach must then decide how many different versions of the same service to support as well as how long to support each version.

There is no right amount of time for a version of a service to be available; for services that undergo constant change, a shorter life cycle might be appropriate, while the same service at different QoS levels might drive support for multiple versions for years. The appropriate life cycle for a service version also depends on the service provider's ability to cope with changes as well as the available resources to support multiple services [Lublinsky 2007b]. Regardless of the strategy in use, it is important to ensure that the provider notifies service consumers who depend on deprecated services about changes in availability and support as part of the ongoing communication and coordination policy.

Recommendation 31: Follow a predictable update and deprecation schedule to make change coordination significantly easier.

3.4 Retirement

At the end of their life cycles, services are usually removed so that they are no longer available to consumers. Managing requests for retired services can involve a complex solution. As we explained in Section 2.2.3, a service broker enables the service infrastructure to handle these requests, for example, by responding with notification about retirement or other updates [Brown 2004]. If the service provider is confident that a newer version of the requested service can provide the requested capability at the desired QoS level, then a common approach is to use XSLT to translate older service requests into up-to-date ones [Peltz 2004].

Another possible issue that might arise when retiring services is the existence of rogue services. Rogue services are "unregistered services that cannot be governed" and can have a significant performance impact on a service's infrastructure [Schepers 2008]. In an SOA environment that supports several versions of the same service, someone might forget to completely disable and remove a service when it is retired. If so, the best-case scenario is that no external consumers know about the retired service and attempt to use it. This means that the only performance impact of continuing to host the rogue service would be from any regular maintenance tasks that the service performs.

However, consumers often fail to update to a newer version and continue to use the retired service as though it is still formally available. If this happens, not only will performance unexpectedly degrade due to the unintended execution of the retired service, but also the consumer may start to receive incorrect data or fail to complete tasks because the called service is no longer tied into the rest of the service infrastructure. Such a situation can negatively affect both the consumer and the service provider, from both a financial and a reputation standpoint. Therefore, in multiversion service environments, it is important to be vigilant about ensuring that every version of every service is fully retired when it has reached the end of its life cycle.

Recommendation 29: At the end of a service's life cycle, manage its retirement process and eliminate all references to the service to prevent rogue services.

4 Related Work

There is a large amount of existing versioning guidance for service-oriented systems. However, much of this guidance focuses on specific standards, technologies, and implementation models, such as WS-* standards for web services. For example, Brown and Ellis discuss how change-type theory relates to services and demonstrate the concept using WSDL and UDDI registries [Brown 2004]. Lublinsky explores the differences among versioning, QoS, and encoding policies as well as using standards such as WSDL 2.0, WS-Addressing, and WS-Policy for implementing these policies [Lublinsky 2004]. Lee also covers these topics [Lee 2005]. Evdemon surveys the differences among message and contract versioning, extensibility, compatibility, and degradation issues for web-service implementations using XML, WSDL, and UDDI technologies [Evdemon 2005]. Diephouse examines specific patterns using SOAP namespaces and other artifacts for versioning web services [Diephouse 2007]. Juric and Sasa extend the BPEL language to include the management of versioned web services [Juric 2010]. This is based on an exploration of WSDL and UDDI that was originally presented by Juric and Sasa [Juric 2009b]. The same authors expanded their study to web-service interfaces and presented the resulting BPEL extensions [Juric 2009a]. Poulin provides a detailed discussion of key artifacts to version and proposes a naming scheme for them [Poulin 2006].

Several researchers also describe custom SOA infrastructures that are built specifically to support versioning. Leitner and colleagues present a limited discussion of versioning concepts, including change types and service proxies that make versioning decisions at runtime [Leitner 2008]. They also investigate transparency as a key attribute and propose a custom versioning solution. Similarly, Fang and colleagues offer a brief review of versioning concepts and a custom versioning solution that they implement on both the service provider and the service consumer [Fang 2007].

We must also note the policy efforts of standards organizations related to implementing or controlling the versioning of web services. The World Wide Web Consortium (W3C) provides the beginnings of a semantically annotated WSDL document called SAWSDL, which would give explicit versioning capabilities to WSDL documents [W3C 2007]. OASIS attempts to address the versioning problem in their WSDM-MOWS document, although it is more of an exploratory paper [OASIS 2004]. IBM produced the WSLA specification that offered a way to represent QoS characteristics, but this work was not continued [Ludwig 2003].

Some work includes the topic of versioning under the heading of a larger topic, such as SOA governance. For example, in their larger discussion of SOA governance, Schepers and colleagues devote a section to change management and mention a number of important versioning issues, such as rogue services and the use of registries for controlling versioning [Schepers 2008].

There are also some high-level examinations of the challenges of SOA versioning. Peltz and Anagol-Subbarao discuss specific versioning strategies for XML schemas, provide some naming conventions, and cover service brokers and general service life cycle concerns [Peltz 2004]. Laskey investigates versioning nomenclature (schemes for setting version numbers), compatibility issues, sufficiency issues, opacity issues, and service-description versioning [Laskey 2008]. Lublinsky explores change types, routing issues, semantic messaging, and life-cycle considerations

[Lublinsky 2007b]. Finally, Conradi and Westfechtel examine versioning models in depth from a theoretical standpoint [Conradi 1998].

5 Summary

Creating a cohesive and comprehensive version-control policy as part of the development effort for a service-oriented system is a complex task. From a versioning perspective, the frequently distributed nature of service-oriented systems development means that the control usually exercised by a single SCM group is often distributed to multiple groups within multiple organizations. As a result, all stakeholders must write and agree to policies that govern what to version, how to version it, how to communicate about and coordinate changes, and how to manage the life cycle of changes. By paying close attention to the key artifacts created during the development of a service-oriented system, and by leveraging tools and recommended practices for putting these key artifacts under version control, developers can build a versioning solution that avoids potential conflicts and promotes good interorganizational behavior.

In this report, we provided guidance on the typical issues that a versioning policy for a service-oriented system must address. We also gave specific guidance on implementing and supporting versioning capabilities in technologies that are commonly used for building services. Finally, we highlighted common problems that can occur in a poorly versioned service-oriented system to reinforce the need for comprehensive versioning policies. In the appendix, we summarize our recommendations. We encourage readers to use these to build comprehensive versioning policies tailored to their system context.

Appendix Summary of Recommendations

Table 1 provides a list of our recommendations in this report, along with the topics to which they pertain.

Table 1: Recommended Practices for Artifact Versioning in Service-Oriented Systems

Number	Topic	Recommendation
1	Key Artifacts	Place all WSDL documents under version control.
2	Key Artifacts	Define data types used in service interfaces in separate XML schemas, and place them under version control.
3	Key Artifacts	If the development effort for service-oriented systems includes composite services, consider the documents that control the composition as key artifacts and place them under version control.
4	Key Artifacts	Ensure that the versioning policy contains guidance for how to handle changes in SOA infrastructure components.
5	Key Artifacts	Identify all metadata that is relevant to service consumers, decide how to document it, and place the resulting artifacts under version control.
6	Service Interface Design	If the service provider will support multiple interfaces for a single service, include policies about how long to support each exposed interface. In addition, use a naming convention that indicates that these are all variants of the same interface.
7	Service Interface Design	Develop policies for how long to support multiple versions of the same interface that account for the faster change rates of unique interfaces. Even though overloaded interfaces have lower change rates, place both the interface itself and the schemas that represent operation input types under version control.
8	Service Interface Design	Consistently with standard software development, version all key artifacts for internal use. From the perspective of the external service consumer, version either the entire service or the individual operation interfaces, depending on the needs of the potential consumers. Avoid exposing version information at a level lower than operation, as that is likely to confuse service consumers.
9	Service Interface Design	If a system provides services at different QoS levels, each with a different service interface, place all exposed interfaces under version control. In addition, use a naming convention that indicates that these are all variants of the same interface.
10	Policy Elements	Construct a comprehensive naming scheme, including version creation thresholds, change types, compatibility rules, and a scheme for determining the identification name or number of a new artifact.
11	Policy Elements	Describe backward- and forward-compatibility requirements and goals in the versioning policy, as appropriate to the context. Actively seek backward compatibility in service-oriented systems development, but realize that forward compatibility is much more difficult to ensure and may not be feasible.
12	Policy Elements	Use major and minor version classification to communicate compatibility issues to service consumers. However, do not make it a critical part of a versioning policy in SOA environments because it is not appropriate for all contexts.
13	Policy Elements	Use basic numeric naming schemes, including major/minor designations, for web services. Consider more complex schemes as the consumer base or capabilities increase in size.
14	Technology Strategies	Use WSDL documents as the backbone of any service-versioning strategy, and use the namespace field to differentiate services and interfaces. In more complex environments, extend or annotate the WSDL format to manage extra information.
15	Technology Strategies	Make service design decisions, particularly regarding the use of namespaces, before defining XML-schema versioning policies.
16	Technology Strategies	For composite services, version the composition-control documents and make them version aware. With BPEL, use the extensions described by Juric and colleagues [Juric 2009a]. Otherwise, consult recommended practices for the

		selected business-process-engine technology to enable version awareness.
17	Key Artifacts/ Technology Strategies	Place SLA documents under version control. Either use an ESB infrastructure to provide a standard way of managing SLA concerns, or investigate custom solutions for SLA management.
18	Technology Strategies	Plan the service infrastructure well in advance to avoid significant infrastructure change. If possible, use common open-technology standards to minimize the potential impact of infrastructure change. Architect service-oriented systems in a way that allows the infrastructure to evolve with minimal disruption to services and consumers.
19	Technology Strategies	In all but the most basic service-oriented systems (i.e., those with a small number of services that have well-known consumers), use some form of broker or router to simplify the interface exposed to consumers and enable greater control by the provider.
20	Tool Strategies	Select a VCS that is sufficiently robust to accommodate all the needs of the software development project. To improve productivity, also select an IDE that integrates well with the chosen VCS.
21	Tool Strategies	In all but the most basic service-oriented systems (i.e., those with a small number of services that have well-known consumers), use registries in conjunction with service repositories to store additional service metadata and related artifacts. For larger implementations, use advanced service registry features to inform service consumers of changes and deal with multiple service versions.
22	Technology Strategies	If possible, use open web-service standards not only to support versioning if necessary but also to ensure compatibility with other systems.
23	Technology Strategies	If standard versioning approaches appear to be insufficient for a given service-oriented system, consult current research for ideas about new standards and methods for extending existing standards.
24	Life-Cycle Policy	Align the version-control policy with the organization's testing strategy.
25	Life-Cycle Policy	Use compatibility testing for new versions of a service for both backward and forward compatibility to ensure proper support for consumers.
26	Life-Cycle Policy	Explicitly determine how many versions of a service to support and for how long.
27	Life-Cycle Policy	Release early versions of a service to support testing by service consumers, but manage and name them consistently to differentiate clearly between test versions and production versions of a service.
28	Life-Cycle Policy	When participating in the construction of a multi-organizational service-oriented system, write a codified communication policy about service changes to ensure the smooth evolution of the system.
29	Life-Cycle Policy	Actively provide notification of changes for transparent service interfaces; use passive policies for opaque service interfaces.
30	Life-Cycle Policy	Select update rates based on customer needs and SLAs, but ensure that the procedure can accommodate on-demand changes in critical situations.
31	Life-Cycle Policy	Follow a predictable update and deprecation schedule to make change coordination significantly easier.
32	Life-Cycle Policy	At the end of a service's life cycle, manage its retirement process and eliminate all references to the service to prevent rogue services.

References

URLs are valid as of the publication date of this document.

[Apache Software Foundation 2011]
Apache Software Foundation. *Subversion Version Control System*. http://subversion.apache.org (2011).

[Bianco 2008]
Bianco, P., Lewis, G., & Merson, P. *Service Level Agreements in Service-Oriented Architecture Environments* (CMU/SEI-2008-TN-021). Software Engineering Institute, Carnegie Mellon University, 2008. http://www.sei.cmu.edu/library/abstracts/reports/08tn021.cfm

[Brown 2004]
Brown, K. & Ellis, M. "Best Practices for Web Services Versioning." *developerWorks*. http://www-128.ibm.com/developerworks/webservices/library/ws-version (2004).

[Christensen 2001]
Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium, 2001. http://www.w3.org/TR/wsdl

[CMMI Product Team 2010]
CMMI Product Team. *CMMI for Development, Version 1.3* (CMU/SEI-2010-TR-033). Software Engineering Institute, Carnegie Mellon University, 2010. http://www.sei.cmu.edu/library/abstracts/reports/10tr033.cfm

[Cobban 2004]
Cobban, M. *What Is BPEL and Why Is It so Important to My Business?* SoftCare, 2004. http://www.softcare.com/whitepapers/wp_whatis_bpel.php

[Conradi 1998]
Conradi, R. & Westfechtel, B. "Version Models for Software Configuration Management." *ACM Computing Surveys 30*, 2 (June 1998): 232–282.

[CVS 2011]
CVS. *Concurrent Versions System*. http://savannah.nongnu.org/projects/cvs (2011).

[Diephouse 2007]
Diephouse, D. *XFire User Guide, Versioning*. http://docs.codehaus.org/display/XFIRE/Versioning (2007).

[Eclipse Foundation 2011a]
Eclipse Foundation. *Eclipse Integrated Development Environment*. http://www.eclipse.org (2011).

[Eclipse Foundation 2011b]
Eclipse Foundation. *Eclipse SOA Platform*. http://www.eclipse.org/eclipsesoa (2011).

[Erl 2009]
Erl, T. & Wilhelmsen, H. “SOA Pattern of the Week (#1): Service Façade.” *SOA World Magazine* (February 2009). http://soa.sys-con.com/node/815612

[Evdemon 2005]
Evdemon, J. “Principles of Service Design: Service Versioning.” *MSDN*. http://msdn.microsoft.com/en-us/library/ms954726.aspx (2005).

[Fang 2007]
Fang, R., Chen, Y., Fong, L., Lam, L., Frank, D., Vignola, C., & Du, N. “A Version-Aware Approach for Web Service Client Application Integrated Network Management,” 401–409. *IM '07: 10th IFIP/IEEE International Symposium on Integrated Network Management*. Munich, Germany, May 2007. IEEE Computer Society Press, 2007.

[Fulton 2009]
Fulton, L., Rymer, J. R., An, M., & Czarnecki, M. *The Forrester Wave: Enterprise Service Buses, Q1 2009*. Forrester Research, 2009.

[García-González 2009]
García-González, J. P. “MSDN Service Registry: A Key Piece for Enhancing Reuse in SOA.” *Microsoft Architecture Journal 21* (September 2009). http://msdn.microsoft.com/en-us/architecture/aa699419

[Hagge 2001]
Hagge, D. “Dynamic Discovery and Invocation of Web Services.” *developerWorks*. http://www.ibm.com/developerworks/webservices/library/ws-udax.html (2001).

[Hamano 2011]
Hamano, J. C., et al. *Git: Fast Version Control System*. http://git-scm.com (2011).

[IBM 2011]
IBM. *Rational ClearCase Software Configuration Management System*. http://www-01.ibm.com/software/awdtools/clearcase (2011).

[Juric 2009a]
Juric, M. B., Sasa, A., & Rozman, I. “WS-BPEL Extensions for Versioning.” *Information and Software Technology 51*, 8 (August 2009): 1261–1274.

[Juric 2009b]
Juric, M. B., Sasa, A., Brumen, B., & Rozman, I. “WSDL and UDDI Extensions for Version Support in Web Services.” *Journal of Systems and Software 82*, 8 (August 2009): 1326–1343.

[Juric 2010]
Juric, M. B. & Sasa, A. “Version Management of BPEL Processes in SOA,” 146–147. *Proceedings of the 2010 6th World Congress on Services (SERVICES '10)*. Miami, FL, July 2010. IEEE Computer Society Press, 2010.

[Laskey 2008]
Laskey, K. “Considerations for SOA Versioning,” 333–337. *2008 12th Enterprise Distributed Object Computing Conference Workshops*. Munich, Germany, Sep. 2008. IEEE Computer Society Press, 2008.

[Lee 2005]
Lee, J. “Architecting Industry Standards for Service Orientation.” *MSDN*. http://msdn.microsoft.com/en-us/library/ms978270.aspx (2005).

[Leitner 2008]
Leitner, P., Michlmayr, A., Rosenberg, F., & Dustdar, S. “End-to-End Versioning Support for Web Services,” 59–66. *Proceedings of the 2008 IEEE International Conference on Services Computing, Vol. 1 (SCC '08)*. Honolulu, HI, July 2008. IEEE Computer Society Press, 2008.

[Lewis 2009]
Lewis, G. “Is SOA Being Pushed Beyond Its Limits?” *Microsoft Architecture Journal 21* (September 2009). http://msdn.microsoft.com/en-us/architecture/aa699422

[Lhotka 2007]
Lhotka, R. “A SOA Versioning Covenant.” *SearchWinDevelopment.* http://searchwindevelopment.techtarget.com/tip/A-SOA-versioning-covenant (2007).

[Lublinsky 2004]
Lublinsky, B. “Supporting Policies in Service-Oriented Architecture.” *developerWorks*. http://www.ibm.com/developerworks/webservices/library/ws-support-soa (2004).

[Lublinsky 2007a]
Lublinsky, B. “Defining SOA as an Architectural Style.” *developerWorks*. http://www.ibm.com/developerworks/architecture/library/ar-soastyle (2007).

[Lublinsky 2007b]
Lublinsky, B. “Versioning in SOA.” *Microsoft Architecture Journal 11* (April 2007). http://msdn.microsoft.com/en-us/library/bb491124.aspx

[Ludwig 2003]
Ludwig, H., Keller, A., Dan, A., King, R. P., & Franck, R. *Web Service Level Agreement (WSLA) Language Specification*. IBM Corporation, 2003. http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf

[MacKenzie 2006]
MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., & Metz, R., eds. *Reference Model for Service Oriented Architecture 1.0*. Oasis, 2006. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[NetBeans 2011]
NetBeans Integrated Development Environment. http://netbeans.org (2011).

[OASIS 2004]
OASIS. *Web Services Distributed Management: Management of Web Services (WSDM-MOWS)* (Draft). http://www.oasis-open.org/committees/download.php/5664/wd-wsdm-mowsversioning (2004).

[OASIS 2011]
OASIS. *UDDI*. http://uddi.xml.org (2011).

[Peltz 2004]
Peltz, C. & Anagol-Subbarao, A. "Design Strategies for Web Services Versioning." *SOA World Magazine* (April 2004). http://soa.sys-con.com/node/44356

[Perforce 2011]
Perforce. *Perforce Configuration Management System*. http://www.perforce.com (2011).

[Poulin 2006]
Poulin, M. "Service Versioning for SOA." *SOA World Magazine* (July 2006). http://soa.sys-con.com/node/250503

[Schepers 2008]
Schepers, T. G. J., Iacob, M. E., & Van Eck, P. A. T. "A Lifecycle Approach to SOA Governance," 1055–1061. *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*. Fortaleza, Ceará, Brazil, Mar. 2008. ACM, 2008.

[Schmelzer 2006]
Schmelzer, R. "Solving the Service Granularity Challenge." *ZapThink*. http://www.zapthink.com/2006/03/09/solving-the-service-granularity-challenge (2006).

[Watt 1993]
Watt, D. *Programming Language Concepts and Paradigms*. Prentice Hall PTR, 1993.

[Wikipedia 2011]
Wikipedia. *Software Versioning*. http://en.wikipedia.org/wiki/Software_versioning (2011).

[W3C 2007]
World Wide Web Consortium. *Semantic Annotations for WSDL and XML Schema*. http://www.w3.org/TR/sawsdl (2007).

[Yu 2005]
Yu, L. "Understanding UDDI's tModel." *The Code Project*. http://www.codeproject.com/KB/XML/understandingTModels.aspx (2005).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	January 2012	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Best Practices for Artifact Versioning in Service-Oriented Systems	FA8721-05-C-0003

6. AUTHOR(S)

Marc Novakouski, Grace Lewis, William Anderson, and Jeff Davenport

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2011-TN-009

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ESC/CAA 20 Schilling Circle, Building 1305, 3rd Floor Hanscom AFB, MA 01731-2125	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report describes some of the challenges of software versioning in an SOA environment and provides guidance on how to meet these challenges by following industry guidelines and recommended practices. Managing change in software systems becomes more difficult as the software increases in size, complexity, and dependencies. Part of this task is *software versioning*, in which version identifiers are assigned to software artifacts for the purpose of managing their evolution. However, software versioning is not a self-contained task. Versioning decisions affect a wide range of processes that fall under the broad heading of change management. With the advent of service-oriented architecture (SOA) as a software-development paradigm, software versioning has become even more entwined with the software life cycle, mainly due to the highly distributed nature, multiproduct outcome, and multilayer implementation of service-oriented systems. The report describes typical items that a versioning policy for a service-oriented system should contain, including which artifacts to version, how to apply version control, and the impact of versioning on each phase of the life cycle within an SOA infrastructure.

14. SUBJECT TERMS	15. NUMBER OF PAGES
service-oriented architecture, software versioning, life-cycle management	42

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL